27 Jul 2015 on Ember Services

Ember Services Tutorial

Have you ever wondered how to use a Ember service in your Ember app? I know I have so this tutorial will go over all the things you should know to get up to speed. Let's start with a definition.

What is Ember.Service?

Ember.Service is a class singleton object that holds on to state. It's lazy instantiated when it is used and it is never destroyed as long as the application runs. It isolates responsibilities of the application without using global variables.

When Should You Use It?

Services can be helpful in several situations. Here are a few examples that you might want to use an Ember service.

- **Session Data**
- **APIs that talk to a server**
- **WebSockets**
- **GeoLocation data**
- **Events pushed from a server**

What You'll Learn In This Tutorial

In this post we'll go over Dependency Injection, what it is and how to use it. We'll deep dive into services and give a couple of examples so you'll know what to do if you ever need to use one.

Dependency Injection

You can't really talk about services without talking about dependency injection (DI). DI and service lookup are two important Ember framework concepts. When we talk about DI what we are saying is that we can take objects and inject them into other objects during instantiation. What that means is that we can take our service and inject it into our routes,controllers, templates or components so they can be used.

This is really important for our service. For example if we have an API we'll need to be able to talk to it to retrieve the information we need. For example let's say we need the API in all our components. With DI this as easy as one line in our initializer.

```
// some initializer file
application.inject('component', 'api', 'service:api')
```

If you've ever used Ember-Data then you've used DI. Ember-Data injects the store into all routes and controllers. That's why you can do commands like this.

```
//controller
 this.store.find('item').then(function(items){
        controller.set('items', items);
      });
```

In the next sections we'll see an example of a service working with DI.

Simple Example Using A Service

For the first example we'll create a super simple service. Its only function is to return a string from a method and to keep track of a property. You can follow along with the source code here.

Prereqs

- Ember CLI (and all it's dependencies)
- Node or iojs

As of this writing that's 1.13.1. Please start with these commands.

Setup

```
$ ember new ServiceTest
$ cd ServiceTest
$ ember g service start
$ ember g component comp-test
$ ember g initializer init
```

These commands will generate the scaffolding for our project. Remember that a component must have a hyphen in the name. To learn more about components check out my component example tutorial here.

Service

Let's begin by setting up the service.

```
// app/services/start.js
import Ember from 'ember';

export default Ember.Service.extend({
    isAuthenticated: true,
    thisistest: function() {
        return "this is erik";
    }
});
```

As you can see all we are doing is returning a string and keeping a property name isAuthenticated.

Next up is our component.

Component

```
// app/components/comp-test.js
import Ember from 'ember';
var inject = Ember.inject;

export default Ember.Component.extend({
    start: inject.service(),
    message: 'test',
    actions: {
        pressMe: function() {
            var testText = this.get('start').thisistest();
            this.set('message',testText);
            console.log(this.get('start').isAuthenticated);
        }
    }

});
```

From the top you can see we created a new variable called inject. This is just so I don't have to type *Ember.inject* every time. The *start: inject.service()* is one way of injecting the start service into this component. We'll go over the other way in a second.

Keep in mind you can omit the service name if the property name matches the service name. In the above example the property name is start which matches the service we created earlier. If it didn't match we would have to do something like this.

```
othername: inject.service('start'),
...
```

We could then use *othername* instead in our code.

In the component we have an action called *pressMe* which udpates the message property to the text returned from the service method *thisistest*. For good measure I also log the isAuthenticated into the console.

Initializer

The alternative way of injecting is using the initializer. Like this.

```
// app/initializers/init.js
export function initialize(container, app) {
          app.inject('component', 'start', 'service:start');
}

export default {
  name: 'init',
  initialize: initialize
};
```

This initializer will inject the start service into every component with the name start. In this case we don't need to include the inject.service into the component. Either way this should work.

Component Template

Now for a little housekeeping we'll add a really simple component

template.

```
// app/templates/components/comp-test.hbs
<button {{action "pressMe"}}>push me</button><br>
{{message}}
```

The button action is bound to the button and the message property will be updated.

Application Template

Finally we'll take a look at the application template.

```
// app/templates/application.hbs
<h2 id="title">Welcome to Service Ember.js</h2>

{{outlet}}
{{comp-test}}
```

What It Should Look Like

In the above code I added the comp-test component to the template. If all goes well it should look like this.

Ember Socket Service Example

I wrote earlier that a good example of using a service is when you're dealing with WebSockets. WebSockets are an easy way to send information from the client to the server.

I explored Ember and WebSockets in an earlier tutorial. In that tutorial I used the most excellent EmberJS WebSockets addon by Travis Hoover. It did a lot of the heavy work and the API was really easy to use. The addon created a service so I didn't have to create one.

For this example I'll be using SockJS and we'll be building our own service. We'll create a node server as well. If you like to follow along you can check out the Ember code here and the node code here.

The application is a basic chat room. Multiple clients can connect, pick a username and send to all other users connected. To keep it simple the chat data isn't saved anywhere, although it wouldn't be too difficult to add a Redis backend.

Node Code

We'll begin by checking out the server code.

```
$ npm init
$ npm install sockjs --save
```

After running npm init you can just press enter 10 times to get through all the prompts. For this tutorial the only dependency we care about is sockjs.

```
// app.js
var http = require('http');
var sockjs = require('sockjs');


var clients = {};

// Broadcast to all clients
function broadcast(message){
  // iterate through each client in clients object
  for (var client in clients){
    // send the message to that client
    clients[client].write(message);
  }
}

var echo = sockjs.createServer({ sockjs_url: 'http://cdn.jsdelivr
echo.on('connection', function(conn) {
    clients[conn.id] = conn;

    conn.on('data', function(message) {
        console.log('received ' + message);
        broadcast(message);
    });
    conn.write("hello from the server thanks for connectiong!");
    conn.on('close', function() {
        delete clients[conn.id];
        });
    console.log("connected");
});

var server = http.createServer();
```

```
echo.installHandlers(server, {prefix:'/echo'});
server.listen(7000, '0.0.0.0');
```

I'm not going to lie I stole most of this code directly from the sample application on the SockJS github page. I did a few modifications so it sent data to all connected clients. I won't get into too much detail on this however the program essentially echos any message it receives to all other clients.

To start the server we simply run this command.

```
$ node app.js
```

Emberjs Code

As with the last example we'll create all these files.

```
$ ember new ServiceSockJS
$ cd ServiceSockJS
$ ember g service sockjs
$ ember g component chat-room
$ ember g initializer init
$ bower install sockjs
```

This time I called my component chat-room. It will talk to the service and update the template.

Configuration

To use SockJS we'll need to use the client library in Ember. Since SockJS

is a Non-AMD asset we'll need to import it a certain way for it to work.

Instead of using the app.import in our Broccoli file and then importing it in our application we just need to app.import it.

```
// ember-cli-build.js"
...
 app.import('bower_components/sockjs/sockjs.min.js');
...
```

If we double check the SockJS documentation we'll see that SockJS is a global variable and we can use it anywhere in our application. Be aware that JSHint, a program built into Ember CLI that helps detect errors, will complain anytime you use SockJS in your application. An easy way of fixing this is adding a *global SockJS* comment to the top of any files that we use SockJS in. We'll go into that in a little bit later.

Another needed, albeit somewhat annoying feature of Ember CLI is the Ember CLI Content security Policy. This is not really used in production and is used as more of a reminder to developers to keep CSP and security in mind when developing Ember applications.

For now we'll just add these lines to the environment.js file.

```
// config/environment.js
...
APP: {
      // Here you can pass flags/options to your application inst
      // when it is created
    },
```

```
            contentSecurityPolicy: {
               'default-src': "'none'",
               'script-src': "'self' 'unsafe-inline' 'unsafe-eval'
               'font-src': "'self'",
               'connect-src': "'self' ws://localhost:7000 localhos
               'img-src': "'self'",
               'report-uri':"'localhost'",
               'style-src': "'self' 'unsafe-inline'",
               'frame-src': "'none'"
            }
...
```

This will remove all the warnings we have when dealing with our Ember application.

Service

First we'll take a peek at the sockjs service file.

```
// app/services/sockjs.js
/* global SockJS */
import Ember from 'ember';
var run = Ember.run;
var socket;

export default Ember.Service.extend(Ember.Evented,{
    setupSockjs: function() {
        socket = new SockJS('http://localhost:7000/echo');
        socket.addEventListener('message', run.bind(this, functio
            this.trigger('messageReceived', event.data);
            console.log(event.data);
```

```
        }));
    }.on('init'),
    sendInfo: function(message) {
        socket.send(message);
        console.log(socket);
    }

});
```

Lot's to look at here so we'll break it down.

```
/* global SockJS */
```

This is the comment that we need at the top of the file so JSHint won't complain about the SockJS global variable.

```
export default Ember.Service.extend(Ember.Evented,{
});
```

We want to have this service use the built in Ember.Evented mixin. The neat thing about this mixin is that it allows the creation of events that can be subscribed and emitted. We'll need this later in the code so we can use *trigger*

```
setupSockjs: function() {
}.on('init')
```

This method setupSockjs is where the SockJS socket will be setup and

the event listener will be created. The *on('init')* is an observer that will fire after initialization of the object.

```
socket = new SockJS('http://localhost:7000/echo');
```

This line is simple. We just create a new variable with SockJS. The URL should match the local server you have running. *echo* was the prefix created earlier in the app.js file.

```
socket.addEventListener('message', run.bind(this, function(event)
            this.trigger('messageReceived', event.data);
            console.log(event.data);
        }));
```

The *addEventListener* will fire when messages come in. This is a built in method for SockJS.

Next is *run.bind*. I wrote a tutorial about the Ember run loop earlier this year and I briefly talked about Ember.run. We have to use this so that Ember can keep track of all our request correctly in the run loop.

The *this.trigger* creates a new event called messsageReceived. Trigger is apart of the Event.Evented class. The event created will also pass the event.data object. This is important since later we'll subscribe to *messageReceived* so we can display it in our template.

Just so we know it's working I'll log the event.data to the web browser console. This should just be the text that is received from the server.

The last thing we need to take a look at is the sendInfo method.

```
    sendInfo: function(message) {
        socket.send(message);
        console.log(socket);
    }
```

This method accepts a message object that it sends to the socket server. I made socket global so it can be accepted by this method. (Forgive the camelcase in this program, I just like using it)

That's about it for the service.

Component

Here is the code for the component.

```
// app/components/chat-room.js
import Ember from 'ember';

export default Ember.Component.extend({
    message: '',

    setup: function() {
           this.get('sockjs').on('messageReceived',this, 'message
    }.on('init'),

    messageReceived: function(message){
        $('#chat-content').val(function(i, text){
            return text + message+ '\n';
        });
        this.set('message',message);
```

```
    },
    actions: {
        enter: function(info,username) {
            var send = this.get('sockjs');
            send.sendInfo(username + ': ' + info);

        }
    }

});
```

We'll break this down.

```
    setup: function() {
            this.get('sockjs').on('messageReceived',this, 'message
    }.on('init'),
```

This is another setup function that fires on init. Remember the event we created earlier in the service? Well now we can subscribe to it by using *on*. The first parameter is the name of the event, the second parameters is the binding, and the last is the callback to execute. Essentially anytime this event is triggered the messageReceived method will be executed.

Here is our callback.

```
messageReceived: function(message){
        this.$('#chat-content').val(function(i, text){
            return text + message+ '\n';
        });
        this.set('message',message);
```

```
    },
```

When a message is received we'll do a simple append to the message with a newline so it appears correctly in our text box. We'll go over the chat-room template soon and this will make more sense.

```
actions: {
        enter: function(info,username) {
            var send = this.get('sockjs');
            send.sendInfo(username + ': ' + info);

        }
    }
```

This action is triggered when the user submits the chat data. We send to the service the message we want sent to the service.

Chat-room template file

```
// app/templates/components/chat-room.hbs
    <textarea id="chat-content" style="width:500px;height:300px"
    {{input type='text' placeholder='User Name' value=uname}}
    {{input type='text' placeholder='Chat Message' value=mess}}
    <button {{action 'enter' mess uname}}>Send</button>

Message received:{{message}}
```

This template file is fairly simple. We have a big text area that will display the chat info. We use two Ember input helpers that will help us keep track

of the user name and the message we want to send.

The enter action is sent and the two properties are sent with it.

Initializer

```
export function initialize( container, application ) {
   application.inject('component', 'sockjs', 'service:sockjs');
}

export default {
  name: 'websockets',
  initialize: initialize
};
```

We only have one component but just for fun I injected my sockjs service to every component, in case we need to use it in the future.

Code Application

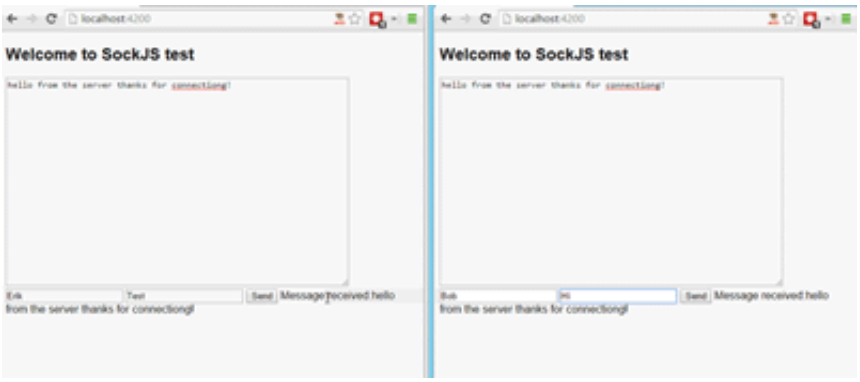
```
// app/templates/application.hbs
h2 id="title">Welcome to SockJS test</h2>

{{chat-room}}
```

In this file I just added the component we just created to it.

What does it look like?

I'll see if I can put up a demo of it somewhere but for now you'll have to be satisfied with some animated gifs. You can of course run the code yourself and give it a whirl. Let me know if you have any issues.



Conclusion

So today we went over dependency injection. How to do it on individual files or globally. Then we went over services, why you should use them and a few examples. The first example went was simple. The second was a chatroom.

Image Credit Linde Gas

Comment? Tweet me at @ErikCH

If you like this tutorial signup for my mailing list! I'll give you free stuff!

Are you a software developer?

Join me and receive a FREE Ember testing cheat sheet and a FREE Aurelia getting started cheat sheet. As well as advice and information on the latest in Node.js, Ember CLI and JavaScript.

email address

Join Now

I hate SPAM. I won't share your your email address with anyone, ever.

If you like to reach me find me on Twitter or Google+.

10 Comments Program With Erik Shun

Recommend 1 Share Sort by Best

Join the discussion…

Marcellin Nshimiyimana · 5 months ago

Thanks for this great post! I am assuming Ember services run on top of web workers

1 · Reply · Share ›

Troy S ➚ Marcellin Nshimiyimana · 5 months ago

No, they are plain objects. However you could totally encapsulate a web worker if you wanted to.

1 · Reply · Share ›

selvagsz · 5 months ago

Thanks for the write up @erik. Just one note
""It is only instantiated once when the application loads"" - I believe services are lazily instantiated when they are consumed.

1 · Reply · Share ›

Erik Mod ➚ selvagsz · 5 months ago

That sounds right, I'll update the tutorial! Thanks!

1 · Reply · Share ›

krishna · 2 months ago

that's a nice tutorial to get some idea about services. But when I tried to implement the tutorial I got this error

Serving on http://localhost:4200/

2015-11-10 14:25 ember[17162] (FSEvents.framework) FSEventStreamStart: register_with_server: ERROR: f2d_register_rpc() => (null) (-21)

events.js:85

throw er; // Unhandled 'error' event

throw er; // Unhandled 'error' event

^

Error: watch EMFILE

at exports._errnoException (util.js:746:11)

at FSEvent.FSWatcher._handle.onchange (fs.js:1161:26)

Reply · Share ›

Erik Mod ➚ krishna · 2 months ago

What version of Ember are you using? Do you have the source code up somewhere? The version I have up on my github page works.

Reply · Share ›

James Dixon · 3 months ago

Thanks for the great article, Erik!

You gave some examples of when you'd use a service. Is there any reason not to use a service for something that isn't necessarily holding state? For example, what about a service that returns a list of US States or Countries? Or in general, something that I need to use in multiple places? I realize that this stuff could be returned by a utility function or that is also injected, but I can't see a reason why i wouldn't just use a service.

Reply · Share ›

Erik Mod ➚ James Dixon · 2 months ago

You don't want to over use services. The reasoning is that it can lead to brittle interdependent code. You can use it in the case your mentioning, just be careful.

Reply · Share ›

Juan Carlos Quintero · 5 months ago

Hi **@Erik** thanks for the tutorial! I just want to comment out that i follow all the steps and i came across with an error in the app/services/sockjs.js, the var socket is defined in the line 5, but is redefined in the line 9, and then in the line 16 inside the sendInfo method is not available. After that everything works perfect with all the steps, again thanks! Waiting for the full Ember course..!

Reply · Share ›

Erik Mod ➚ Juan Carlos Quintero · 5 months ago

Oops! Thanks I fixed it! Good catch

Reply · Share ›